Clase 6 — 12.12.2025

#JAVA #androidstudio

👨‍🏫 ***Profesor**: Joan Salvador Gordi Ortega*
📘 **Programación Multimedia y Dispositivos Móviles**
🗓️ **Clase 6**— 12/12/2025
🎯 **Tema:** Primer Proyecto Android Studio Continuación del desarrollo

---

1️⃣ Estructura general de un proyecto Android

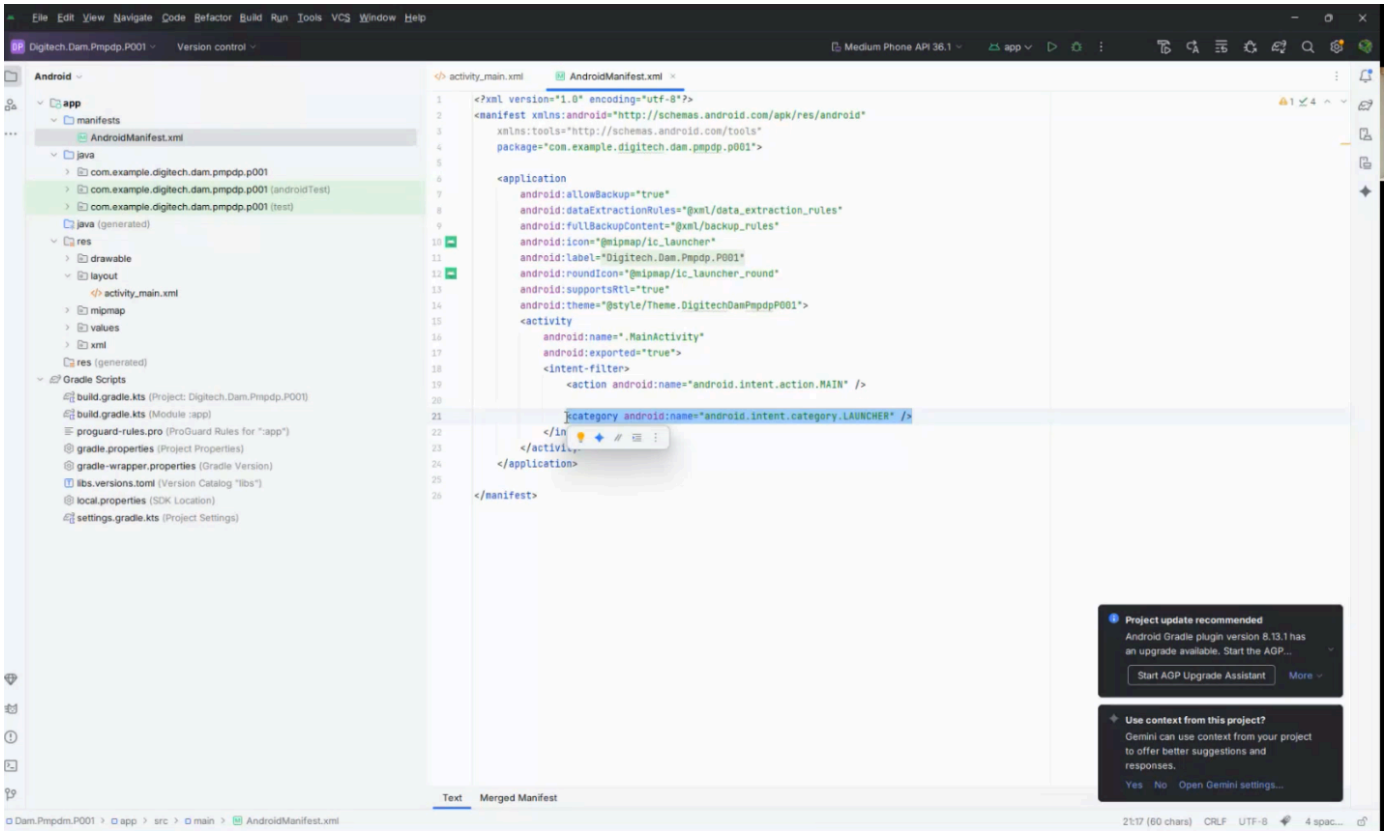
Un proyecto Android sigue una **estructura fija y estandarizada**. La idea es separar claramente:

- **Configuración global** (manifest + Gradle)
- **Lógica** (Java/Kotlin)
- **Recursos** (layouts, imágenes, textos, estilos)

Si entiendes esta estructura, Android Studio deja de parecer un laberinto… y pasa a ser un laberinto **con mapa**.

📂 1.1 Carpetas principales (vista “Android”)

En la vista típica de Android Studio verás (simplificado):



- **manifests**
  - `AndroidManifest.xml`
- **java**
  - Paquetes con clases (`MainActivity` , etc.)
- **res**

- `layout/` (pantallas XML)
- `drawable/` (imágenes y recursos gráficos)
- `mipmap/` (iconos de la app)
- `values/` (strings, colores, estilos)

> Nota: aunque la vista “Android” lo agrupa bonito, físicamente existe `app/src/main/...`

1.2 Carpeta `manifests`

📄 `AndroidManifest.xml`

Archivo **obligatorio** del proyecto Android. Define aspectos globales de la aplicación:

- Nombre del paquete (identificador)
- Componentes declarados (Activities, Services, Receivers)
- Actividad principal (entrada)
- Permisos
- Tema, iconos y configuración de la app

📌 **Concepto clave:**

> Aquí se define **qué pantalla se abre al lanzar la app** y qué “piezas” existen oficialmente.

🔎 Ejemplo de Manifest típico (completo y realista)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.digitechfp.dam.pmpdp.p002.saludo">

    <!-- Permisos (solo si se necesitan) -->
    <!-- <uses-permission android:name="android.permission.INTERNET" /> -->

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:label="P002 Saludo"
        android:supportsRtl="true"
        android:theme="@style/Theme.DamPmpdpP002Saludo"
        tools:targetApi="31">

        <activity
            android:name=".MainActivity"
            android:exported="true">

            <!-- Activity principal -->
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

        </activity>
```

```
    </application>

</manifest>
```

🔹 Qué significan las partes importantes

- `<manifest ... package="...">`
  - Identifica el paquete base del proyecto.
  - Hoy en día, parte del “namespace” también se controla desde Gradle, pero el manifest sigue siendo esencial.
- `<application ...>`
  - Configura la app completa: iconos, etiqueta (nombre visible), tema, backups, etc.
- `<activity android:name=".MainActivity">`
  - Declara una pantalla.
  - Si una Activity no está declarada (o no se usa el sistema de “activity aliases”), Android no la puede lanzar.
- `android:exported="true"`
  - Indica si esa Activity puede ser lanzada por componentes externos.
  - En Android 12+ es obligatorio definirlo cuando existe un `intent-filter`.

1.3 Actividad principal (`MAIN` + `LAUNCHER`)

Dentro del `manifest` se encuentra el `intent-filter`:

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

- `MAIN` → marca el **punto de entrada**
- `LAUNCHER` → hace que aparezca en el lanzador (icono del móvil)
- Solo **una Activity** debería tener ambos (la “principal”)

🧠 Traducción humana: *“Android, empieza por aquí”*.

2️⃣ Concepto de Activity

Una **Activity** representa una **pantalla** de la aplicación.

🔹 Qué hace una Activity

- Dibuja una interfaz (a través de un layout)
- Gestiona eventos (clics, textos, etc.)
- Sigue un **ciclo de vida** (Android puede pausarla, destruirla, recrearla)

📌 En este proyecto:

- `MainActivity.java` → lógica
- `activity_main.xml` → interfaz

🔁 Mini mapa del ciclo de vida (lo mínimo para ubicarte)

- `onCreate()` → se crea la pantalla (inicialización)
- `onStart()` → pasa a ser visible
- `onResume()` → lista para interactuar
- `onPause()` → se va a segundo plano
- `onStop()` → ya no es visible
- `onDestroy()` → se destruye

📌 Para esta práctica, el foco está en `onCreate()`.

3️⃣ Separación de responsabilidades

Android sigue el principio:

> **La lógica va en Java, la interfaz en XML**

Esto ayuda a:

- Reutilizar layouts
- Mantener el código más limpio
- Diseñar pantallas sin tocar la lógica

📂 3.1 Carpeta `java`

Contiene las **clases Java**:

- `MainActivity.java`
- Otras Activities
- Clases auxiliares
- Gestión de eventos (listeners)

Ejemplo (estructura mínima):

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

📂 3.2 Carpeta `res/layout`

Contiene los **layouts XML**:

- `activity_main.xml`
- Define qué se ve
- No debería contener lógica

📌 El nombre `activity_main` suele corresponder con `MainActivity` por convención.

📂 3.3 Carpeta `res/values`

Muy importante aunque al principio se ignore:

- `strings.xml` → textos (mejor que hardcodearlos en XML)
- `colors.xml` → colores
- `themes.xml` / `styles.xml` → estilos y temas

Ejemplo en `strings.xml`:

```
<resources>
    <string name="app_name">P002 Saludo</string>
    <string name="titulo">Introduce tu nombre:</string>
    <string name="hint_nombre">Escribe aquí</string>
    <string name="btn_saludar">Saludar</string>
</resources>
```

4️⃣ Relación entre Activity y Layout

En `MainActivity.java`:

```
setContentView(R.layout.activity_main);
```

Esta línea:

- Carga el XML
- "Infla" el layout y lo convierte en vista real
- Permite que luego puedas hacer `findViewById()`

Si falta → pantalla en blanco.

🧩 Ejemplo más completo (lo típico de clase)

```
public class MainActivity extends AppCompatActivity {

    private EditText etNombre;
    private Button btnSaludar;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        etNombre = findViewById(R.id.etNombre);
        btnSaludar = findViewById(R.id.btnSaludar);
    }
}
```

📌 Importante:

- `setContentView()` debe ir **antes** de `findViewById()`.

5️⃣ Archivo `activity_main.xml`

Define la **composición visual** de la pantalla.

📐 5.1 Layout raíz: `ConstraintLayout`

Se utiliza normalmente:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

</androidx.constraintlayout.widget.ConstraintLayout>
```

🔹 Para qué sirven esos `xmlns`

- `xmlns:android` → atributos estándar (`android:text`, `android:id`...)
- `xmlns:app` → atributos de librerías (por ejemplo, constraints)
- `xmlns:tools` → solo para diseño/preview (no afecta en ejecución)

✅ Ventajas de ConstraintLayout

- Flexible para distintos tamaños
- Permite posicionar con restricciones (top/bottom/start/end)
- Evita anidar muchos layouts (mejor rendimiento)

6️⃣ Componente `TextView`

Elemento visual para **mostrar texto**.

🧾 Ejemplo básico

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hola Mundo" />
```

🧾 Ejemplo realista con `id`, tamaño y constraints

```
<TextView
    android:id="@+id/tvTitulo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/titulo"
    android:textSize="18sp"
    android:layout_marginBottom="16dp"

    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toTopOf="@id/etNombre" />
```

🔹 Propiedades importantes

- `android:id="@+id/..."`
  - Crea un identificador para luego poder usar `findViewById()`.
- `layout_width / layout_height`
  - `wrap_content` → se ajusta al contenido
  - `match_parent` → ocupa todo el espacio disponible
- `android:text`
  - Texto a mostrar. Recomendado: `@string/...`.
- `android:textSize`
  - Unidades en `sp` (escala con accesibilidad del usuario).
- `android:layout_margin...`
  - Separación externa (margen).
- `app:layout_constraint...`
  - Posicionamiento dentro de `ConstraintLayout`.

📌 Diferencia útil:

- **margin** → espacio fuera del componente
- **padding** → espacio dentro del componente

Ejemplo con `padding`:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/titulo"
    android:padding="8dp" />
```

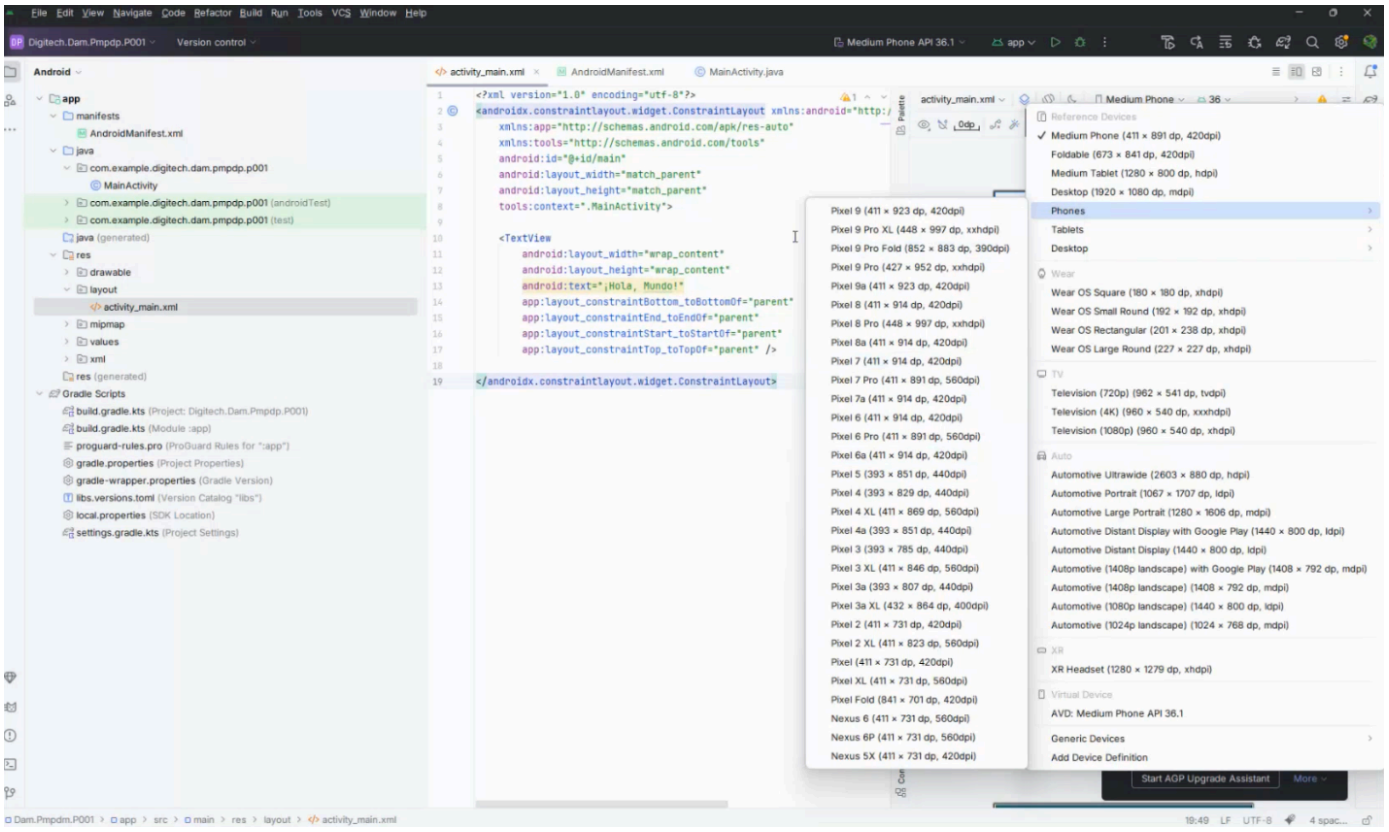
7️⃣ Previsualización del layout

Android Studio ofrece una **vista previa en tiempo real** del XML.

🔍 Opciones de visualización

Desde el desplegable:

- Diferentes móviles
- Tablets
- Escritorio
- Wear OS
- Automotive



Esto permite:

- Ver cómo se adapta el diseño
- Detectar problemas de escalado

📌 No todos los móviles son iguales. Android tampoco.

8️⃣ Device Manager (Dispositivos virtuales)

Acceso desde:

```
Tools → Device Manager
```

Permite:

- Crear dispositivos virtuales (AVD)
- Elegir tamaño de pantalla
- Elegir versión de Android (API)
- Probar la app sin móvil físico

📌 Recomendación:

> Probar siempre en **varios tamaños y APIs**.

9️⃣ Ejecución de la aplicación

Para ejecutar la app:

- Botón ▶ `Run 'app'`
- Atajo: **Mayús + F11**

La aplicación se carga en:

- El dispositivo virtual activo
- O un dispositivo físico conectado

🔁 1️⃣0️⃣ Flujo de trabajo recomendado

1. Ejecutar la app una vez (Run)
2. Realizar cambios en XML o Java
3. Usar:

```
Apply Changes and Restart Activity
```

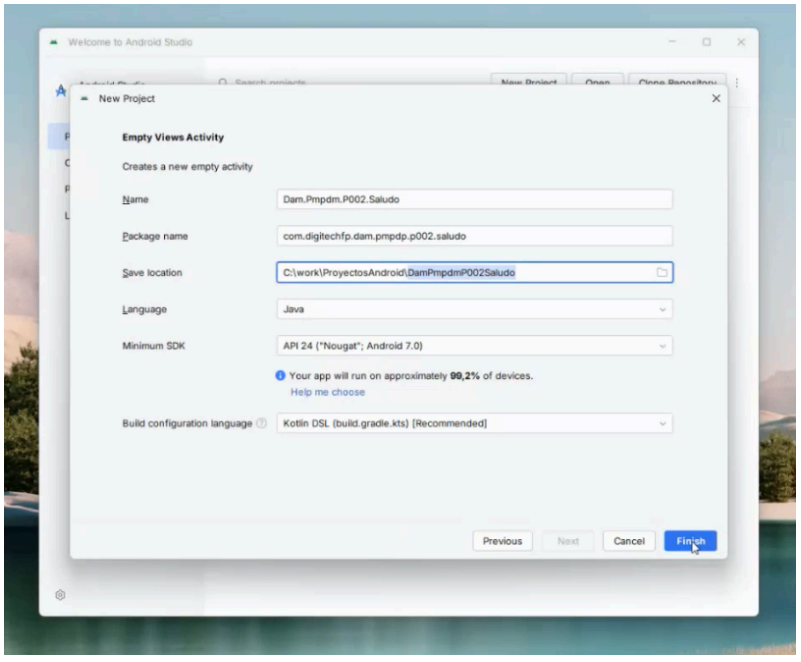
Ventajas:

- Más rápido
- No reinicia todo el emulador
- Ideal para iterar diseño

1️⃣1️⃣ Nuevo proyecto: App de saludo

Se crea un nuevo proyecto:

- **Plantilla:** Empty Views Activity
- **Nombre:** Dam.Pmpdm.P002.Saludo
- **Lenguaje:** Java
- **SDK mínimo:** API 24 (Android 7.0)



📌 Esta plantilla genera:

- Activity básica
- Layout vacío
- Estructura estándar

1️⃣2️⃣ `MainActivity.java`

Características:

- Extiende de `AppCompatActivity`
- Método principal: `onCreate()`

```
public class MainActivity extends AppCompatActivity
```

📌 `onCreate()`:

- Se ejecuta al crear la pantalla
- Inicializa la interfaz
- Punto de arranque lógico

1️⃣3️⃣ Componentes interactivos

✏️ `EditText`

Permite al usuario **introducir texto**.

🔘 `Button`

Permite ejecutar acciones mediante clic.

Para usarlos en Java es obligatorio:

```
import android.widget.EditText;
import android.widget.Button;
```

Estas variables representan **referencias en memoria** a elementos visuales.

1️⃣4️⃣ Vinculación XML ↔ Java

Cada componente tiene un `id` en XML:

```
android:id="@+id/etNombre"
```

En Java se vincula con:

```
etNombre = findViewById(R.id.etNombre);
```

📌 Esto conecta:

- El objeto Java
- Con el elemento visual real

Sin esta línea → el componente no existe para Java.

1️⃣5️⃣ Evento click del botón (`Button`)

Ahora se define **qué ocurre cuando el usuario pulsa el botón**. Este comportamiento es equivalente al manejo de eventos en **Java Swing**, pero adaptado al modelo Android.

🔘 15.1 `setOnClickListener()`

Para reaccionar a un clic se utiliza un **listener**:

```
btnSaludar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // lógica al hacer click
    }
});
```

📌 Claves importantes:

- `setOnClickListener()` asocia un evento al botón
- `new View.OnClickListener()` crea una **clase anónima**
- `onClick(View v)` se ejecuta **cada vez que se pulsa el botón**

🧠 Traducción humana: *"Cuando alguien pulse esto, haz lo siguiente"*.

1️⃣6️⃣ Recuperar el texto del `EditText`

Dentro de `onClick()` se obtiene el texto introducido por el usuario:

```
String nombre = etNombre.getText().toString();
```

🔹 Qué hace cada parte

- `etNombre` → referencia al `EditText`
- `getText()` → obtiene el contenido
- `toString()` → lo convierte en `String`

📌 Sin `toString()` no podemos trabajar cómodamente con el texto.

1️⃣7️⃣ Validación del contenido

Antes de usar el texto, se comprueba que **no esté vacío**:

```
if (!nombre.isEmpty()) {
    // continuar
}
```

Esto evita:

- Mensajes sin nombre
- Errores de lógica
- UX pobre (apps que saludan al vacío existencial)

1️⃣8️⃣ Construcción del mensaje

Si el texto es válido, se crea el mensaje:

```
String mensaje = "Hola " + nombre;
```

📌 Concatenación simple de cadenas:

- Texto fijo (`Hola`)
- Texto dinámico (nombre del usuario)

1️⃣9️⃣ Mostrar mensaje con `Toast`

Para mostrar información rápida en pantalla:

```
Toast.makeText(MainActivity.this, mensaje, Toast.LENGTH_SHORT).show();
```

🔹 Explicación por partes

- `MainActivity.this` → contexto actual
- `mensaje` → texto a mostrar
- `Toast.LENGTH_SHORT` → duración
- `.show()` → ejecuta el Toast

📌 `Toast` :

- No bloquea la app
- Desaparece solo
- Ideal para mensajes breves

2️⃣0️⃣ Prueba y ejecución en dispositivos

Para comprobar que la app funciona:

1. Activar un **device virtual o físico**
2. Pulsar ▶ `Run 'app'`
3. Esperar a que la app se cargue

📌 Importante:

> Si se cambia de dispositivo, **hay que volver a ejecutar la app**.

El profesor detecta que:

- La app no se actualiza si no se hace `Run`
- Android Studio puede parecer bloqueado, pero es el device

2️⃣1️⃣ Previsualización del layout

Android Studio permite **previsualizar en tiempo real** el diseño definido en los archivos XML de la carpeta `res/layout` .

Esta previsualización **no ejecuta la app**, sino que renderiza el XML para que podamos ver cómo quedaría la pantalla.

📌 Es una herramienta de **diseño y comprobación visual**, no de ejecución.

🔍 21.1 Opciones de visualización

En la parte superior derecha del editor de layouts encontramos un **desplegable de dispositivos** que permite cambiar la previsualización.

Desde este menú podemos simular:

- 📱 **Diferentes móviles** (Pixel, Nexus, tamaños pequeños y grandes)
- 📲 **Tablets**
- 🖥️ **Escritorio**
- ⌚ **Wear OS**
- 🚗 **Automotive**
- 📺 **Televisión**

Cada opción cambia:

- Resolución
- Densidad de píxeles (dpi)
- Proporción de pantalla

📌 El XML **no cambia**, lo que cambia es **cómo se interpreta**.

🧠 21.2 ¿Por qué es importante?

La previsualización permite:

- Ver si los elementos:
  - Se solapan
  - Se salen de pantalla
  - Quedan demasiado juntos o separados
- Detectar problemas de **escalado**
- Comprobar que `ConstraintLayout` está bien definido

📌 Idea clave:

> No todos los móviles son iguales. Android tampoco.

2️⃣2️⃣ Device Manager (Dispositivos virtuales)

El **Device Manager** permite crear y gestionar **dispositivos virtuales (AVD)** para ejecutar la aplicación.

Acceso desde:

```
Tools → Device Manager
```

📱 22.1 ¿Qué es un AVD?

Un **AVD (Android Virtual Device)** es una simulación completa de un dispositivo Android:

- Sistema operativo Android real
- Hardware simulado
- Pantalla, botones, sensores, etc.

📌 Es como tener “un móvil dentro del ordenador”.

⚙️ 22.2 Qué podemos configurar

Al crear un dispositivo virtual podemos elegir:

- 📐 **Tamaño de pantalla**
- 📏 **Resolución**
- 🤖 **Versión de Android (API)**
- 🔄 Orientación (portrait / landscape)
- 🧠 Arquitectura (x86, arm…)

Ejemplo:

- Pixel 9
- Android 14 (API 34)
- Pantalla grande
- Alta densidad

📌 22.3 Buenas prácticas

Recomendación del profesor:

> Probar siempre en **varios tamaños y APIs**.

Motivos:

- Una app puede verse bien en un Pixel grande y mal en un móvil pequeño
- Versiones antiguas pueden comportarse distinto
- Ayuda a detectar errores antes de entregar

2️⃣3️⃣ Ejecución de la aplicación

Para **ejecutar realmente** la app (no solo verla):

▶️ Opciones de ejecución

- Botón ▶ **Run 'app'**
- Atajo de teclado: **Mayús + F11**

Antes de ejecutar:

- Debe haber **un dispositivo activo** (virtual o físico)

📲 23.1 ¿Dónde se ejecuta la app?

La app se carga en:

- El **dispositivo virtual activo**
- O un **dispositivo físico conectado por USB**

📌 Si no hay ningún dispositivo activo → la app no arranca.

🔁 2️⃣4️⃣ Flujo de trabajo recomendado

Una vez la app ya se ha ejecutado **al menos una vez**, Android Studio permite un flujo más rápido.

🔄 Flujo típico

1. Ejecutar la app con `Run`
2. Realizar cambios en:
   - XML (diseño)
   - Java (lógica)
3. Usar la opción:

```
Apply Changes and Restart Activity
```

⚡ 24.1 Ventajas de este flujo

- Mucho más rápido
- No reinicia todo el emulador
- Mantiene el dispositivo encendido
- Ideal para iterar diseño y pruebas pequeñas

📌 Importante:

> Si **cambias de dispositivo**, debes volver a hacer **Run**.

El profesor tuvo problemas en clase porque:

- Cambió de device
- La app no se cargaba
- El problema no era el código, sino que **no se había relanzado la app**

2️⃣5️⃣ Nuevo proyecto: App de saludo

Se crea un nuevo proyecto para empezar desde cero.

📋 Configuración usada

- **Plantilla:** Empty Views Activity
- **Nombre:** `Dam.Pmpdm.P002.Saludo`
- **Lenguaje:** Java
- **SDK mínimo:** API 24 (Android 7.0)

🧱 25.1 ¿Qué genera esta plantilla?

Android Studio crea automáticamente:

- Una `MainActivity`
- Un layout `activity_main.xml`
- Un `AndroidManifest.xml` configurado
- Estructura de carpetas estándar
- Archivos Gradle necesarios

📌 Es una base limpia, perfecta para aprender y practicar.

🧠 25.2 Por qué se usa `Empty Views Activity`

- No añade código extra
- No usa Compose
- Facilita entender:
  - Relación Activity ↔ Layout
  - Eventos
  - Componentes básicos (`TextView` , `EditText` , `Button`)

Ideal para **primeros proyectos**.

2️⃣6️⃣ `MainActivity.java`

`MainActivity` es la **clase principal** de la aplicación y representa la **pantalla activa** que el usuario ve al abrir la app.

📌 Características clave

- Es una **Activity**
- Extiende de `AppCompatActivity`
- Gestiona:
  - Ciclo de vida
  - Lógica
  - Eventos de usuario
  - Comunicación con el layout XML

🧱 26.1 Declaración de la clase

```
public class MainActivity extends AppCompatActivity {
```

Esto implica:

- Android reconoce esta clase como una pantalla
- Hereda métodos esenciales (`onCreate` , `onPause` , etc.)
- Puede cargar layouts y manejar eventos

📌 Si no extiende de `AppCompatActivity` , **no puede funcionar como Activity estándar**.

🔁 26.2 Método `onCreate()`

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

Qué ocurre paso a paso:

1. Android crea la Activity
2. Llama automáticamente a `onCreate()`

3. `super.onCreate()` inicializa la parte interna del sistema
4. `setContentView()`:
    - Carga el XML
    - Dibuja la interfaz
    - Permite acceder a los componentes con `findViewById()`

📌 Regla de oro:

> **Sin `setContentView()` no hay interfaz.**

2️⃣7️⃣ Componentes interactivos

Una app empieza a ser **interactiva** cuando el usuario puede **introducir datos** y **realizar acciones**.

En este proyecto se usan dos componentes clave:

✏️ 27.1 `EditText`

Permite al usuario **escribir texto**.

Usos habituales:

- Nombre
- Usuario
- Email
- Contraseña (con `inputType`)

Ejemplo típico en XML:

```
<EditText
    android:id="@+id/etNombre"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Escribe aquí"
    android:inputType="textPersonName" />
```

📌 `hint`:

- Texto de ayuda
- Desaparece al escribir

🔘 27.2 `Button`

Permite **ejecutar una acción** cuando el usuario pulsa.

Ejemplo en XML:

```
<Button
    android:id="@+id/btnSaludar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Saludar" />
```

📌 Un botón **no hace nada por sí solo**.
Necesita lógica en Java.

📦 27.3 Importación de clases

Para usar estos componentes en Java:

```
import android.widget.EditText;
import android.widget.Button;
```

Sin estos imports:

- El compilador no reconoce los componentes
- El proyecto no compila

📌 Importante:

> Estas variables son **referencias en memoria**, no el componente visual en sí.

2️⃣8️⃣ Vinculación XML ↔ Java

Para que Java pueda manipular un componente del layout, debe **vincularlo** usando su `id`.

🔹 28.1 Identificador en XML

```
android:id="@+id/etNombre"
```

- `@+id` → crea el identificador
- `etNombre` → nombre único dentro del layout

🔹 28.2 Vinculación en Java

```
etNombre = findViewById(R.id.etNombre);
btnSaludar = findViewById(R.id.btnSaludar);
```

Qué ocurre aquí:

1. Android busca en el layout cargado
2. Encuentra el componente con ese ID
3. Devuelve una referencia
4. Se asigna a la variable Java

📌 Sin esta vinculación:

- Java **no conoce** el componente
- No se puede leer ni modificar su contenido

📌 Regla crítica:

> `findViewById()` **siempre después** de `setContentView()`.

2️⃣9️⃣ Evento click del botón (`Button`)

Aquí se define **qué ocurre cuando el usuario pulsa el botón**.

Android usa un modelo basado en **listeners**, similar a Java Swing, pero adaptado al framework Android.

🔘 29.1 `setOnClickListener()`

```
btnSaludar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // código que se ejecuta al hacer click
    }
});
```

Qué está pasando:

- `setOnClickListener()`:
  - Registra un escuchador
- `new View.OnClickListener()`:
  - Clase anónima
- `onClick(View v)`:
  - Método que Android ejecuta automáticamente al pulsar

📌 No se llama manualmente a `onClick()`.
Android lo invoca cuando detecta el evento.

🧠 Traducción humana:

> "Cuando se pulse este botón, ejecuta este código".

3️⃣0️⃣ Recuperar el texto del `EditText`

Dentro de `onClick()` se obtiene el texto introducido por el usuario.

```
String nombre = etNombre.getText().toString();
```

🔍 Desglose

- `etNombre` → referencia al `EditText`
- `getText()` → devuelve un objeto `Editable`
- `toString()` → convierte el contenido en `String`

📌 Es obligatorio convertir a `String` para:

- Comparar
- Validar
- Concatenar
- Mostrar por pantalla

3️⃣1️⃣ Validación del contenido

Antes de usar el texto introducido por el usuario es **imprescindible validarlo**.
En este caso, se comprueba que el nombre **no esté vacío**.

🔎 Validación básica

```
if (!nombre.isEmpty()) {
    // continuar con la lógica
}
```

Qué está pasando aquí

- `nombre` es un `String` obtenido del `EditText`
- `isEmpty()` devuelve:
  - `true` → cadena vacía (`""`)
  - `false` → hay contenido

Al usar `!nombre.isEmpty()` estamos diciendo:

> "Solo continúa si el usuario ha escrito algo".

🧠 Por qué es necesaria esta validación

Evita varios problemas comunes:

- ❌ Mensajes del tipo: `Hola`
- ❌ Comportamientos incoherentes
- ❌ Mala experiencia de usuario (UX)
- ❌ Lógica poco profesional

📌 En apps reales, **siempre se valida la entrada del usuario**, aunque sea algo simple.

🔁 Variante habitual (más defensiva)

En proyectos más avanzados es habitual combinar varias comprobaciones:

```
if (nombre != null && !nombre.trim().isEmpty()) {
    // continuar
}
```

- `trim()` elimina espacios en blanco
- Evita que `" "` pase como nombre válido

3️⃣2️⃣ Construcción del mensaje

Una vez validado el texto, se construye el mensaje que se mostrará al usuario.

🧩 Concatenación de cadenas

```
String mensaje = "Hola " + nombre;
```

Qué ocurre aquí

- `"Hola "` → texto fijo
- `nombre` → texto dinámico introducido por el usuario
- El operador `+` concatena ambas cadenas

Resultado ejemplo:

- Entrada: `Carlos`
- Resultado: `Hola Carlos`

📌 Esta concatenación es simple, clara y suficiente para este ejercicio.

🔁 Alternativa con `String.format()` (opcional)

```
String mensaje = String.format("Hola %s", nombre);
```

Más habitual en:

- Proyectos grandes
- Internacionalización
- Textos complejos

3️⃣3️⃣ Mostrar mensaje con `Toast`

Para mostrar el mensaje al usuario se utiliza un **Toast**, un aviso flotante típico de Android.

📢 Uso básico de `Toast`

```
Toast.makeText(MainActivity.this, mensaje, Toast.LENGTH_SHORT).show();
```

🔍 Explicación por partes

- `MainActivity.this`
  - Contexto actual
  - Indica **desde dónde** se lanza el mensaje
- `mensaje`
  - Texto que se va a mostrar
- `Toast.LENGTH_SHORT`
  - Duración corta (~2 segundos)
  - Alternativa: `Toast.LENGTH_LONG`
- `.show()`
  - Ejecuta y muestra el Toast
  - Sin esta llamada, **no aparece nada**

📌 Error típico:

> Crear el `Toast` pero olvidar `.show()`.

🧠 Características de `Toast`

- ✔ No bloquea la app
- ✔ No requiere interacción
- ✔ Desaparece automáticamente
- ✔ Ideal para mensajes breves y feedback rápido

📌 No es adecuado para:

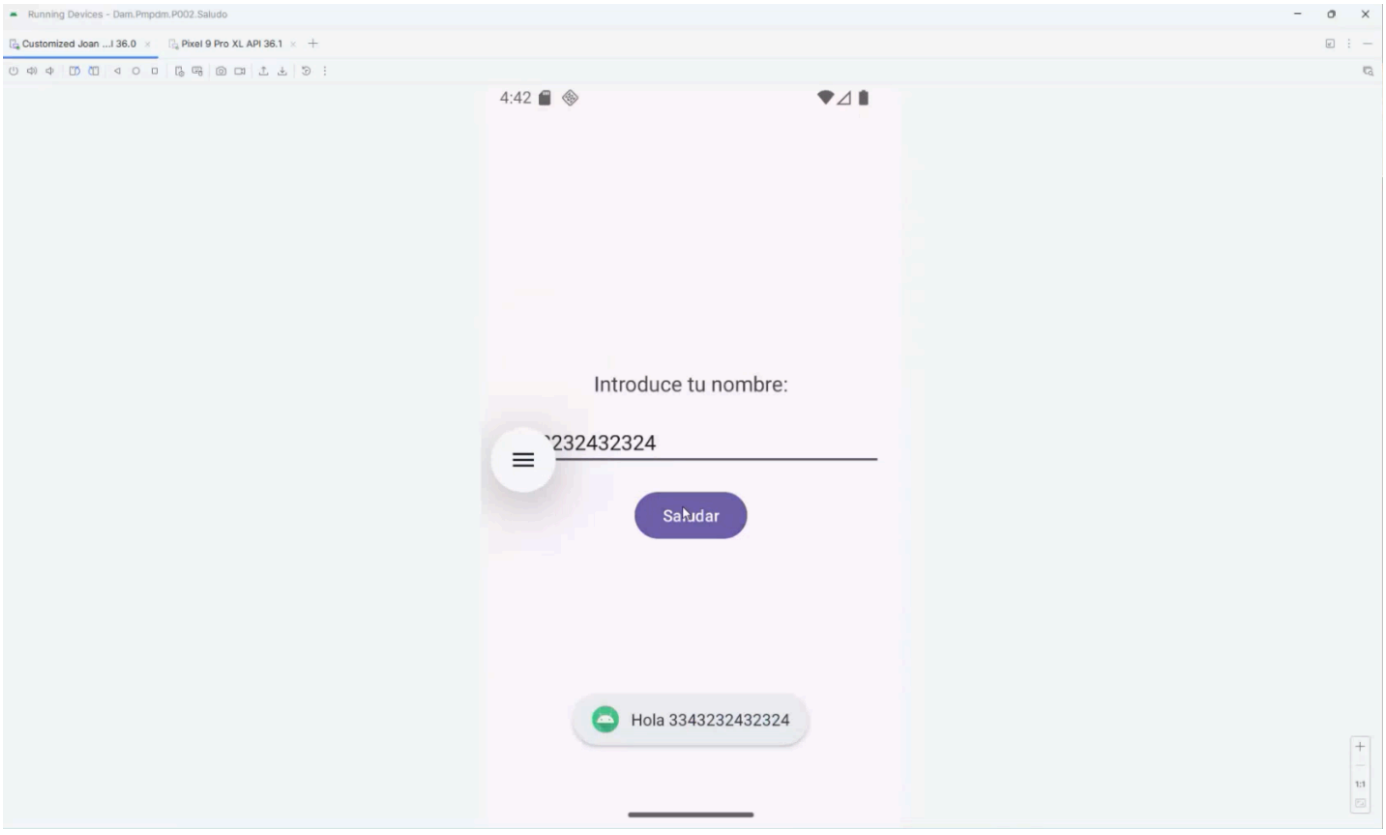
- Mensajes largos
- Confirmaciones críticas
- Errores complejos

3️⃣4️⃣ Prueba y ejecución en dispositivos

Una vez implementada la lógica, es necesario **probar la aplicación en ejecución real**.

▶️ Pasos para comprobar que funciona

1. Activar un **dispositivo virtual (AVD)** o conectar uno físico
2. Pulsar ▶ **Run 'app'**
3. Esperar a que Android Studio compile y cargue la app



⚠️ Problema detectado en clase (muy importante)

El profesor detecta que:

- Cambiar de dispositivo **no relanza la app automáticamente**
- Android Studio puede parecer bloqueado
- El error **no está en el código**, sino en el flujo de ejecución

📌 Regla clave:

> Si se cambia de device, **hay que volver a hacer** `Run` .

3️⃣5️⃣ Resumen del flujo completo

Este bloque resume **todo el comportamiento funcional** de la app de saludo.

🔁 Flujo paso a paso

1. El usuario introduce su nombre en el `EditText`
2. Pulsa el botón `Saludar`
3. El `OnClickListener` captura el evento
4. Se lee el texto del `EditText`
5. Se valida que no esté vacío
6. Se construye el mensaje
7. Se muestra con `Toast`

📌 **Resultado final**:
Primera app Android **funcional, interactiva y correctamente estructurada**.
